Software Transactional Memory for Multicore OCaml

https://vraajkum.github.io/ocaml-stm-webpage/

Edward Brownhill ebrownhi@andrew.cmu.edu
Vishant Raajkumar vraajkum@andrew.cmu.edu

1 Revised Schedule

Date	Goals	Owner
Apr. 14 – Apr. 17	EP-STM core: tvar type, undo log, in-place writes.	Vishant
	Per-operation conflict check and exponential backoff contention manager.	Edward
Apr. 17 – Apr. 21	Sequential correctness on shared counter.	Edward
	Begin transactional linked list.	Vishant
Apr. 21 – Apr. 24	Linearizability testing with `multicoretests` on counter.	Edward
	Linearizability testing with `multicoretests` on linked list.	Vishant
Apr. 24 – Apr. 26	Transactional hash map implementation.	Both
	Baseline benchmark infrastructure vs. Domainslib `Mutex` and `kcas`.	Both
Apr. 26 – Apr. 28	Full benchmark matrix (3 structures × 3 contention levels (90x50x10) × 1–8 domains).	Both
	Analyze crossover points; draft results section.	Edward
Apr. 28 – Apr. 30	Write final report.	Edward
	Prepare poster.	Vishant
Apr. 30	**Final report due (11:59pm).**	
May 1	**Poster session.** 8:30–11:30am.	

Table 1: Revised project schedule.

2 Progress

Our first major work item was a thorough study of kcas, the only production STM for Multicore OCaml, which serves as our lazy+optimistic reference implementation. Rather than treating it as a black box, we read through the full source to understand how it actually works at the level of atomic operations and memory.

The most important thing we learned is how kcas represents shared locations. Instead of storing a plain value, each location holds a state descriptor with three fields: a before value, an after value, and a which pointer. The which field either marks the location as settled (pointing to a resolved status) or marks it as owned by a specific in-flight transaction. This means any domain that reads a location can determine the correct current value just by following which, without needing a lock.

We also traced through how transactions are recorded and committed. Accesses are logged in a splay tree keyed by location ID, and each logged access is classified as either a read-only CMP (no memory write needed) or a write CAS (owned by this transaction). At commit time, the protocol runs in four phases, install, verify, finish, and release, with a single CAS on the transaction descriptor serving as the linearization point. A key revelation we had is the helping mechanism: if one domain finds a location already owned by another transaction, it completes that transaction itself rather than waiting. This is what gives kcas its lock-freedom guarantee.

This analysis gave us a clear picture of where kcas is fast and where it isn't, and directly shaped the design decisions for our EP-STM.

Then, we started working on our EP-STM implementation. So far, we have implemented the basic project structure setup using Ocaml and Dune. We have also begun writing parts of the implementation such as the atomic global clock. Progress has been a bit slower than expected since we have had to take time to learn the atomic library for Ocaml and study and understand how software transactional memory works.

3 Goals and Deliverables

Work completed:

- Deep study of `kcas` source and the algorithm: state descriptor representation, splay tree transaction log, CMP vs. CAS classification, four-phase commit protocol, and helping mechanism.
- Planned EP-STM design decisions informed by `kcas` study: representation, undo log structure, lock ownership model.
- Informed start to EP-STM implementation.

Revised goals for poster session:

- A correct EP-STM implementation (eager versioning, pessimistic conflict detection) with exponential backoff contention manager, verified with `multicoretests`, a testing library provided by the Ocaml dev team.
- Transactional shared counter, singly-linked list, and hash map implemented on top of EP-STM's primitives, serving as both correctness benchmarks under `multicoretests` and the workloads for performance evaluation.
- Benchmark suite comparing EP-STM against `kcas` and Domainslib `Mutex` across 1–8 domains under low, medium, and high contention.

Nice to haves (if time permits):

- Object-level vs. element-level conflict detection granularity comparison on the hash map benchmark.
- Contention manager study: aggressive abort vs. randomized backoff vs. priority-by-age.

4 Poster Session

We plan to show throughput graphs (transactions/second) across a handful domains for each of the three data structures under low, medium, and high contention, comparing EP-STM, `kcas`, and Domainslib `Mutex`. The key result we hope to demonstrate is the crossover point at which EP-STM's early conflict detection outperforms `kcas`'s deferred detection under high contention.

If a slideshow is a valid poster session format, I think we'd greatly benefit from that. Is it possible to use that in addition to a poster?

5 Concerns

Our primary concern is timeline: the EP-STM implementation is behind the original schedule and must be completed and verified within the next week before benchmarking can begin. The two risks we are watching are (1) correctness under OCaml 5's relaxed memory model, where documentation has wared us that a misplaced fence in the commit path can produce bugs that are extremely difficult to reproduce, and (2) garbage collector interference in benchmark results, which may obscure the true performance characteristics of each design.